ECE 495

Computer Engineering Design Laboratory

Lab 7

Microprogrammed CPU Design

Group 301

Jeremy McLynch
Raghav Bharathan

December 16, 2025

On my honor, I pledge that I have not violated the provisions of the NJIT Academic Honor Code.

	Grade
11/20 Part 1A, 1B ok Demo: 11/25 Part 2 ok Verified: 12/11 Pt 1 ok	
Report:	

[illegible]
Student Signature

[illegible]
Student Signature

Objective and Introduction

The objective of this lab was to take everything we've learned from the previous labs to implement an entire microprogrammed CPU design. The design involved several steps, starting with testing the microsequencer, creating the RTL, sketching a block diagram, completing the uROM table, and finally implementing it using VHDL with LPM components and structural modeling. The CPU was implemented using the Altera DE2 FPGA, and we performed several tests using the provided RAM files. Along with testing the design on the DE2 board, we also created waveform tests that allowed us to understand the timing of each register much better. The CPU takes 16 instructions with seven main register groups that are detailed in the Theory and Methods section.

Components/Software used:

- Altera Cycle IV EP4CE115F29C7 FPGA device with DE2-115 7-segment display, LEDs, and Switches
- Altera Quartus Lite 13.1 (Web Edition)
- USB Cable to USB Blaster connection on DE2 Board

Theory and Methods

The CPU design uses seven main register groups:

1) Program Counter (PC) - 8 bits
2) Memory Address Register (MAR) - 8 bits
3) Instruction Register (IR) - 8 bits
4) Memory Data Register (MDR) - 8 bits
5) Register File (R0/R1) - 8 bits
6) Stack Pointer (SP) - 8 bits
7) Conditional Flip-Flop (Z) - 1 bit

There are 16 instructions, and they can be either 1 byte or 2 bytes. The first byte is the opcode, and if present, the 2nd byte is the address or data.

We first started the design by understanding how the provided microsequencer works, then designing the RTL for each microinstruction by consulting previous examples on the slide. One key detail to note with the RTL design is that we used a downward-growing stack pointer that would move to numerically lower addresses as new data is pushed.

Then, we started designing the block diagram that would serve as a layout for all the components, registers, and MUXs that we would use for the lab. For the block diagram, we labeled both the internal ports and external signals/wires that would connect to those ports, which made it easier for us to do the port mapping when we got to the VHDL implementation.

We also included bus lines, clock lines, and a memory register that would all communicate with each other. Our ALU was relatively simple, consisting of a simple adder of type lpm_add_sub. For the PC increment and SP increment/decrement, we used an lpm_counter.

Using the RTL design and block diagrams, we created the microcode for the instruction set. The microcode for our design was horizontally programmed on an Excel, which we later converted to a .mif file suitable for Quartus as a uROM file. We had 43 uROM addresses with a range from 00 to F0, with 20 bits for the control signal, 1 MAP bit, and 8 bits for the next address. The control signals are described below:

1) add_or_sub (0): Determines addition or subtraction for the ALU's adder
2) write_z (1): Enable signal for the Z register
3) test_nz (2): Test select signal for MUX Z - if test_nz = 0, z ← v, else z ← not v
4) ir_load (3): Maps the instruction during FETCH 3
5) mem_write (4): Write enable for the memory register
6) mar_mux_s0 (5): LSB select signal for the MAR MUX
7) mar_mux_s1 (6): MSB select signal for the MAR MUX
8) mar_load (7): Enable signal for the MAR register
9) r_mux_sel (8): Select signal for the R MUX
10) r_enable (9): Enable signal for the R register - r0_enable <= r_enable and not IR(0), r1_enable <= r_enable and IR(0)
11) mdr_mux_sel (10): Select signal for the MDR MUX
12) mdr_load (11): Enable signal for the MDR register
13) pc_load (12): Enable signal for the PC register
14) pc_inc (13): Increment enable for the PC register
15) inc_sp (14): Increment enable for the SP register
16) sp_load (15): Enable signal for the SP register
17) sp_up (16): Add/Sub (1/0) for the SP register
18) jumpz (17): Conditional Jump Flag - loads PC when (jumpz and z) or pc_load
19) im_sel (18): Immediate select bit
20) Mov_sel (19): Move select bit

Lastly, in the VHDL implementation, we instantiated the CPU using a fully structural approach that directly follows the block diagram and microcode design. The design instantiates individual LPM components for all major datapath elements, including registers (lpm_ff), counters (lpm_counter), multiplexers (lpm_mux), memory (lpm_ram_dq), and the ALU (lpm_add_sub). These components are interconnected using internal signals that represent the CPU's buses and control lines.

The microsequencer (f25lab7_useq) reads the opcode from the instruction register and outputs a microinstruction from the uROM. This microinstruction is decoded into control signals that drive the datapath by selecting MUX inputs, enabling register loads, controlling memory access, and configuring ALU operations. The register file (R0/R1), PC, SP, MAR, MDR, IR, and Z flag are all updated based on these control signals on clock edges.

The datapath uses multiplexers to route data between registers, memory, and the ALU, allowing the same hardware to support multiple instructions. Debug outputs and seven-segment display decoders are included to observe internal register values during simulation and hardware testing. Overall, the VHDL implementation directly maps the RTL and microcode design into a modular, clock-driven CPU architecture.

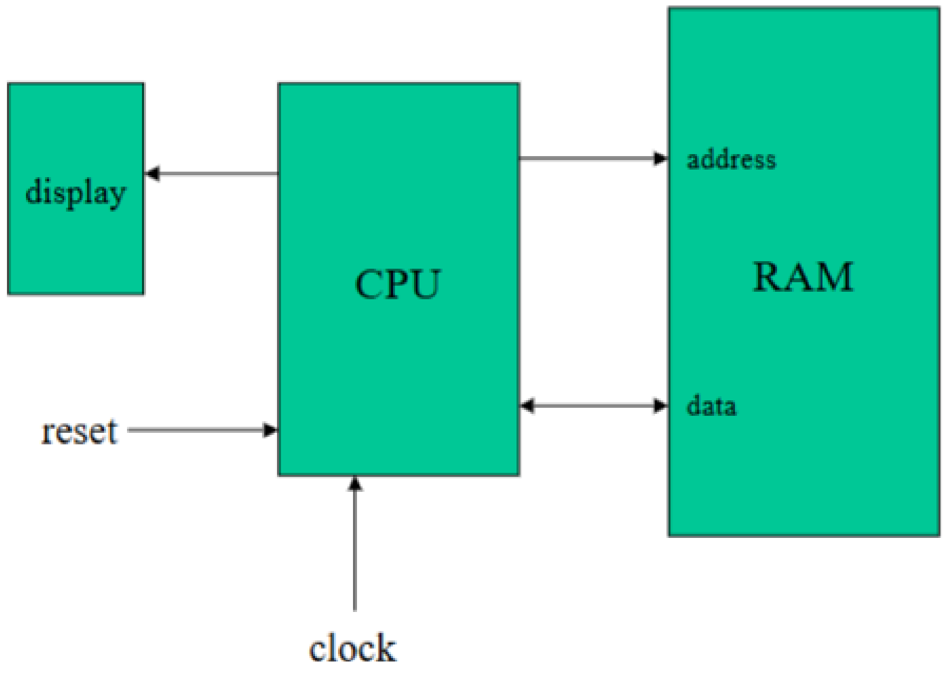
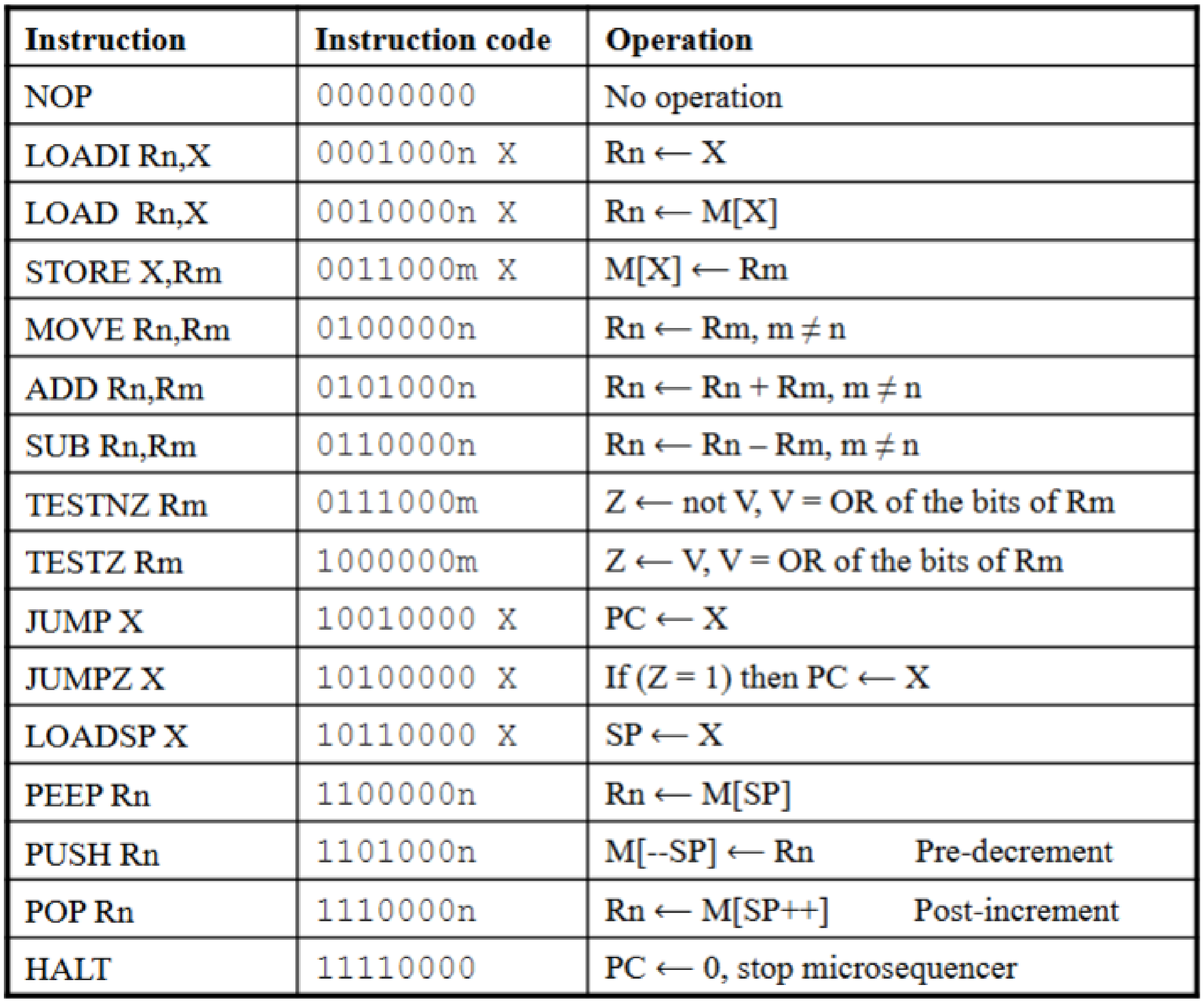


Figure 1. Overarching Diagram [Ref 1. *F25 ECE495 L7.pdf*]



Instruction	Instruction code	Operation
NOP	`00000000`	No operation
LOADI Rn,X	`0001000n X`	Rn ← X
LOAD Rn,X	`0010000n X`	Rn ← M[X]
STORE X,Rm	`0011000m X`	M[X] ← Rm
MOVE Rn,Rm	`0100000n`	Rn ← Rm, m ≠ n
ADD Rn,Rm	`0101000n`	Rn ← Rn + Rm, m ≠ n
SUB Rn,Rm	`0110000n`	Rn ← Rn – Rm, m ≠ n
TESTNZ Rm	`0111000m`	Z ← not V, V = OR of the bits of Rm
TESTZ Rm	`1000000m`	Z ← V, V = OR of the bits of Rm
JUMP X	`10010000 X`	PC ← X
JUMPZ X	`10100000 X`	If (Z = 1) then PC ← X
LOADSP X	`10110000 X`	SP ← X
PEEP Rn	`1100000n`	Rn ← M[SP]
PUSH Rn	`1101000n`	M[--SP] ← Rn Pre-decrement
POP Rn	`1110000n`	Rn ← M[SP++] Post-increment
HALT	`11110000`	PC ← 0, stop microsequencer

Figure 2. CPU Instructions [Ref 1. *F25 ECE495 L7.pdf*]

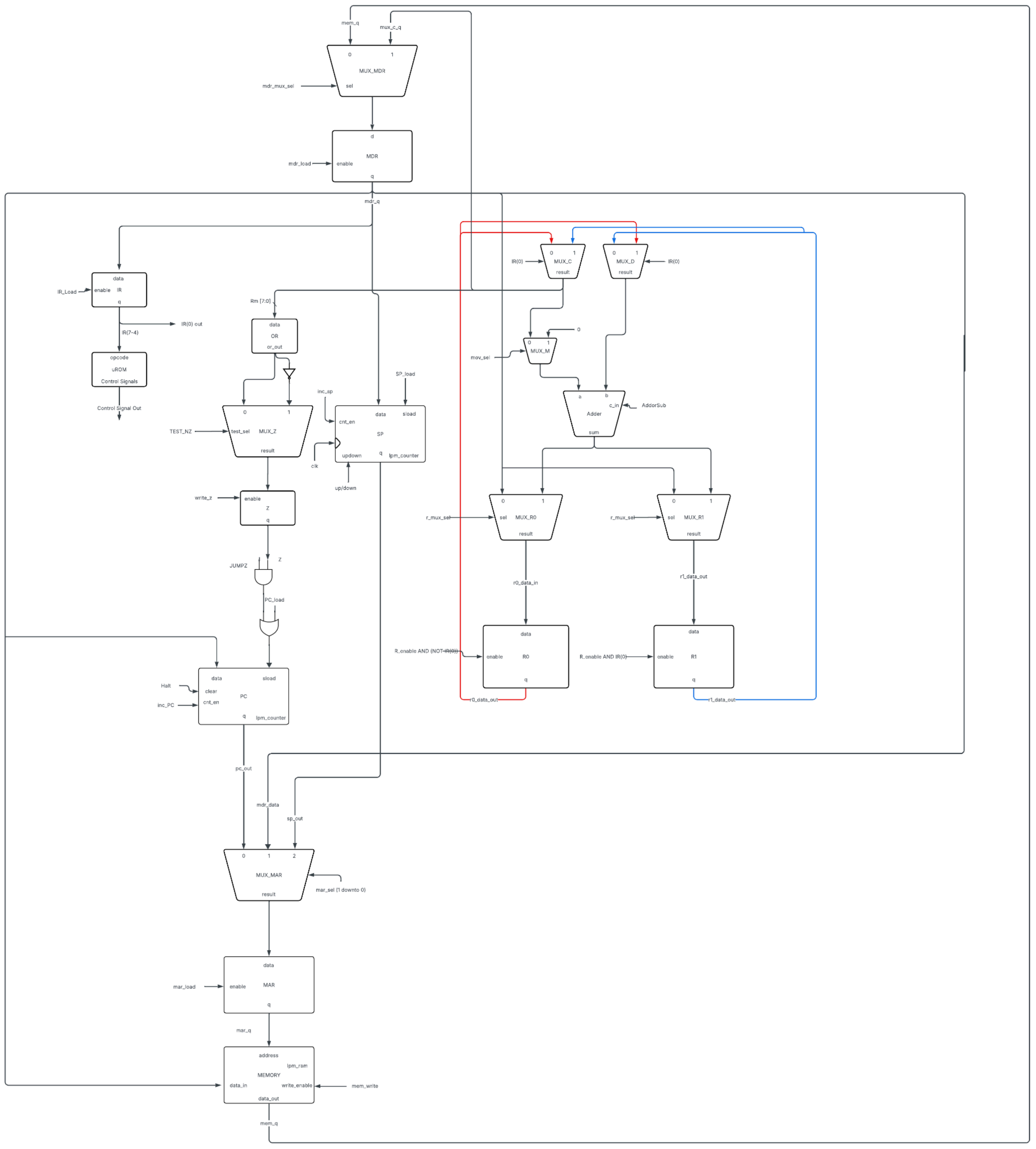


Figure 3. Block Diagram

RTL

FETCH1: MAR ⟵PC
FETCH2: MDR ⟵M[MAR], PC ⟵PC + 1
FETCH3: IR ⟵MDR

NOP
NOP1: (NO OPERATION)

LOADI Rn, X
LOADI1: MAR ⟵ PC
LOADI2: MDR ⟵ M[MAR], PC ⟵ PC+1
LOADI3: R0 ⟵ MDR if IR(0) = 0, R1 ⟵ MDR if IR(0) = 1

LOAD Rn, X
LOAD1: MAR ⟵ PC
LOAD2: MDR ⟵ M[MAR], PC ⟵ PC+1
LOAD3: MAR ⟵ MDR
LOAD4: MDR ⟵ M[MAR]
LOAD5: R0 ⟵ MDR if IR(0) = 0, R1 ⟵ MDR if IR(0) = 1

STORE X, Rm
STORE1: MAR ⟵ *PC*
STORE2: MDR ⟵ *M[MAR], PC ⟵ PC + 1*
STORE3: MAR ⟵ *MDR*
STORE4: MDR ⟵ *R0 if IR(0)=0, MDR ⟵ R1 if IR(0)=1*
STORE5: M[MAR] ⟵ *MDR*

MOVE Rn, Rm
MOVE1: R0 ⟵ *R1 if IR(0)=0, R1 ⟵ R0 if IR(0)=1*

ADD Rn, Rm
ADD1: R0 ⟵ *R0 + R1 if IR(0)=0, R1 ⟵ R1 + R0 if IR(0)=1*

SUB Rn, Rm
SUB1: R0 ⟵ *R0 - R1 if IR(0)=0, R1 ⟵ R1 - R0 if IR(0)=1*

TESTNZ Rm
TESTNZ1: *Z ⟵ not (Rm(0) or Rm(1) or Rm(2) or Rm(3) or Rm(4) or Rm(5) or Rm(6) or Rm(7)), m=IR(0)*

TESTZ Rm
TESTZ1: *Z ⟵ Rm(0) or Rm(1) or Rm(2) or Rm(3) or Rm(4) or Rm(5) or Rm(6) or Rm(7), m=IR(0)*

JUMP X

JUMP1: MAR ⟵ PC

JUMP2: MDR ⟵ M[MAR], PC ⟵ PC +1

JUMP3: PC ⟵ MDR

JUMPZ X

JUMPZ1: MAR ⟵ PC

JUMPZ2: MDR ⟵ M[MAR], PC ⟵ PC+1

JUMPZ3: PC ⟵ MDR if Z = 1

LOADSP X

LOADSP1: *MAR ⟵ PC*

LOADSP2: *MDR ⟵ M[MAR], PC ⟵ PC + 1*

LOADSP3: SP ⟵ *MDR*

PEEP Rn

PEEP1: MAR ⟵ SP

PEEP2: MDR ⟵ M[MAR]

PEEP3: R0 ⟵ MDR if IR(0) = 0, R1 ⟵ MDR if IR(0) = 1

PUSH Rn

PUSH1: SP ⟵ SP - 1

PUSH2: MAR ⟵ SP

PUSH3: MDR ⟵ *R0 if IR(0)=0, MDR ⟵ R1 if IR(0)=1*

PUSH4: M[MAR] ⟵ MDR

POP Rn

POP1: *MAR ⟵ SP*

POP2: MDR ⟵ *M[MAR]*

POP3: *SP ⟵ SP + 1*

POP4: *R0 ⟵ MDR if IR(0)=0, R1 ⟵ MDR im IR(0)=1*

HALT

HALT1: PC ⟵ 0

DE2 Signal Name	**FPGA Pin No.**	**Description**
pcSegHi[0]	PIN_AD18	Seven Segment Digit 5[0]
pcSegHi[1]	PIN_AC18	Seven Segment Digit 5[1]
pcSegHi[2]	PIN_AB18	Seven Segment Digit 5[2]
pcSegHi[3]	PIN_AH19	Seven Segment Digit 5[3]
pcSegHi[4]	PIN_AG19	Seven Segment Digit 5[4]
pcSegHi[5]	PIN_AF18	Seven Segment Digit 5[5]
pcSegHi[6]	PIN_AH18	Seven Segment Digit 5[6]
pcSegLo[0]	PIN_AB19	Seven Segment Digit 4[0]
pcSegLo[1]	PIN_AA19	Seven Segment Digit 4[1]
pcSegLo[2]	PIN_AG21	Seven Segment Digit 4[2]
pcSegLo[3]	PIN_AH21	Seven Segment Digit 4[3]
pcSegLo[4]	PIN_AE19	Seven Segment Digit 4[4]
pcSegLo[5]	PIN_AF19	Seven Segment Digit 4[5]
pcSegLo[6]	PIN_AE18	Seven Segment Digit 4[6]
r0SegHi[0]	PIN_M24	Seven Segment Digit 1[0]
r0SegHi[1]	PIN_Y22	Seven Segment Digit 1[1]
r0SegHi[2]	PIN_W21	Seven Segment Digit 1[2]
r0SegHi[3]	PIN_W22	Seven Segment Digit 1[3]
r0SegHi[4]	PIN_W25	Seven Segment Digit 1[4]
r0SegHi[5]	PIN_U23	Seven Segment Digit 1[5]

r0SegHi[6]	PIN_U24	Seven Segment Digit 1[6]
r0SegLo[0]	PIN_G18	Seven Segment Digit 0[0]
r0SegLo[1]	PIN_F22	Seven Segment Digit 0[1]
r0SegLo[2]	PIN_E17	Seven Segment Digit 0[2]
r0SegLo[3]	PIN_L26	Seven Segment Digit 0[3]
r0SegLo[4]	PIN_L25	Seven Segment Digit 0[4]
r0SegLo[5]	PIN_J22	Seven Segment Digit 0[5]
r0SegLo[6]	PIN_H22	Seven Segment Digit 0[6]
r1SegHi[0]	PIN_V21	Seven Segment Digit 3[0]
r1SegHi[1]	PIN_U21	Seven Segment Digit 3[1]
r1SegHi[2]	PIN_AB20	Seven Segment Digit 3[2]
r1SegHi[3]	PIN_AA21	Seven Segment Digit 3[3]
r1SegHi[4]	PIN_AD24	Seven Segment Digit 3[4]
r1SegHi[5]	PIN_AF23	Seven Segment Digit 3[5]
r1SegHi[6]	PIN_Y19	Seven Segment Digit 3[6]
r1SegLo[0]	PIN_AA25	Seven Segment Digit 2[0]
r1SegLo[1]	PIN_AA26	Seven Segment Digit 2[1]
r1SegLo[2]	PIN_Y25	Seven Segment Digit 2[2]
r1SegLo[3]	PIN_W26	Seven Segment Digit 2[3]
r1SegLo[4]	PIN_Y26	Seven Segment Digit 2[4]
r1SegLo[5]	PIN_W27	Seven Segment Digit 2[5]
r1SegLo[6]	PIN_W28	Seven Segment Digit 2[6]

spSegHi[0]	PIN_AD17	Seven Segment Digit 7[0]
spSegHi[1]	PIN_AE17	Seven Segment Digit 7[1]
spSegHi[2]	PIN_AG17	Seven Segment Digit 7[2]
spSegHi[3]	PIN_AH17	Seven Segment Digit 7[3]
spSegHi[4]	PIN_AF17	Seven Segment Digit 7[4]
spSegHi[5]	PIN_AG18	Seven Segment Digit 7[5]
spSegHi[6]	PIN_AA14	Seven Segment Digit 7[6]
spSegLo[0]	PIN_AA17	Seven Segment Digit 6[0]
spSegLo[1]	PIN_AB16	Seven Segment Digit 6[1]
spSegLo[2]	PIN_AA16	Seven Segment Digit 6[2]
spSegLo[3]	PIN_AB17	Seven Segment Digit 6[3]
spSegLo[4]	PIN_AB15	Seven Segment Digit 6[4]
spSegLo[5]	PIN_AA15	Seven Segment Digit 6[5]
spSegLo[6]	PIN_AC17	Seven Segment Digit 6[6]
zSeg	PIN_F17	LED Green[8]
sw1	PIN_AB28	Slide Switch[0]
sw2	PIN_AC28	Slide Switch[1]

Figure 4. Pin Assignments

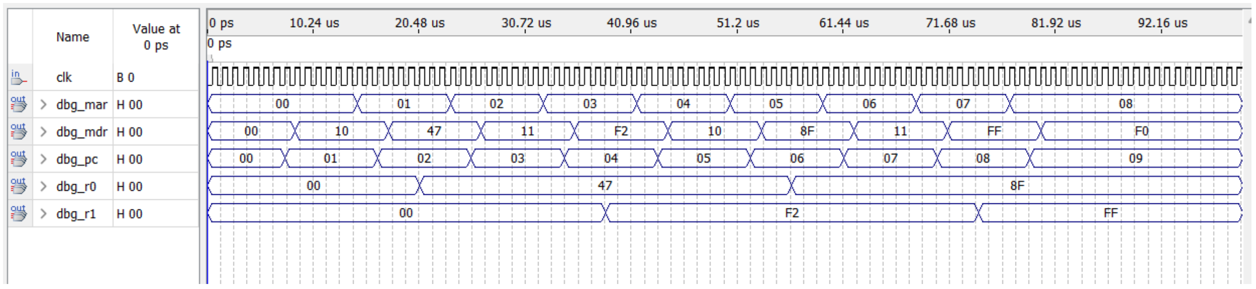


Figure 5. Waveform Test 1 (LOADI)

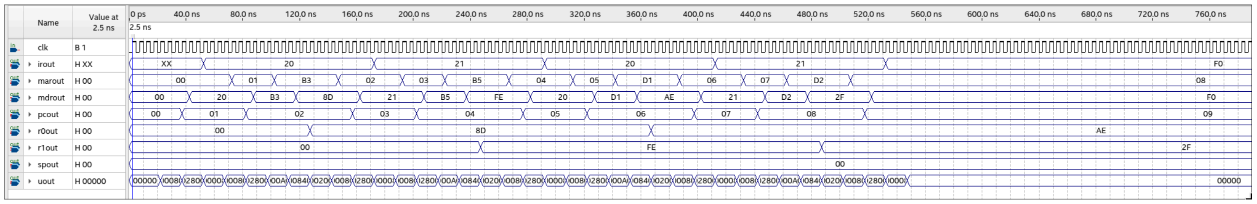


Figure 6. Waveform Test 2 (LOAD)

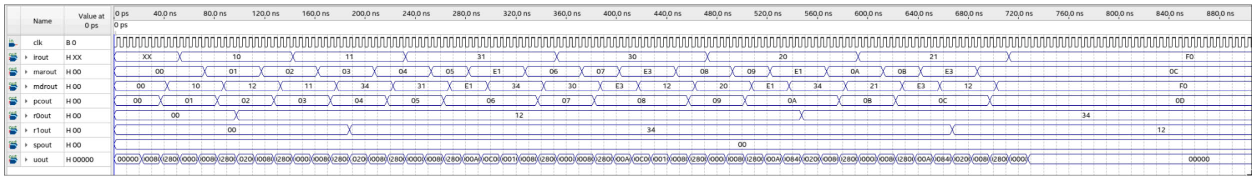


Figure 7. Waveform Test 3 (STORE)

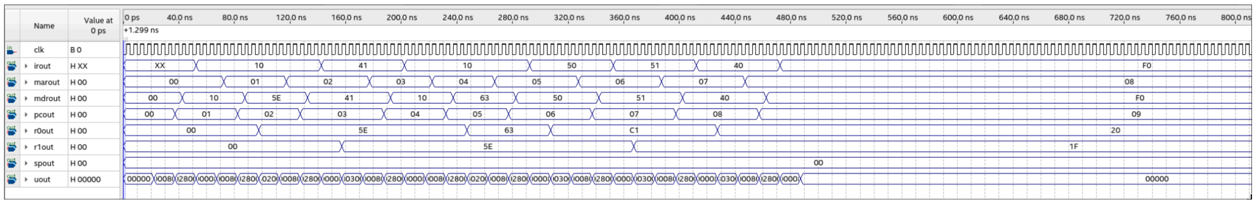


Figure 8. Waveform Test 4 (MOVE & ADD)

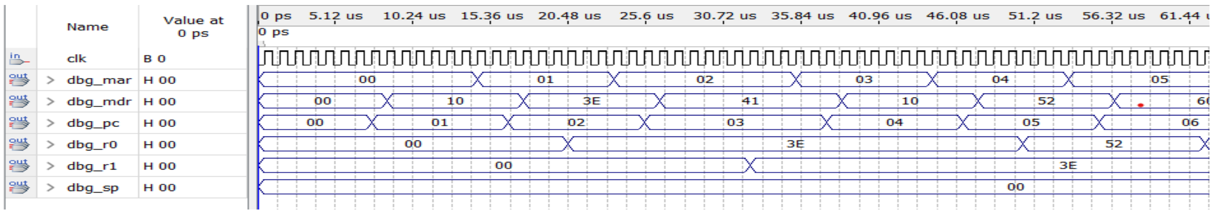


Figure 9. Waveform Test 5 (MOVE & SUB)

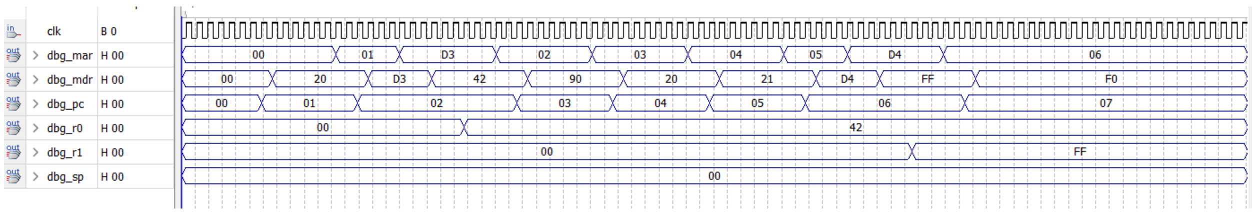


Figure 10. Waveform Test 6 (JUMP)

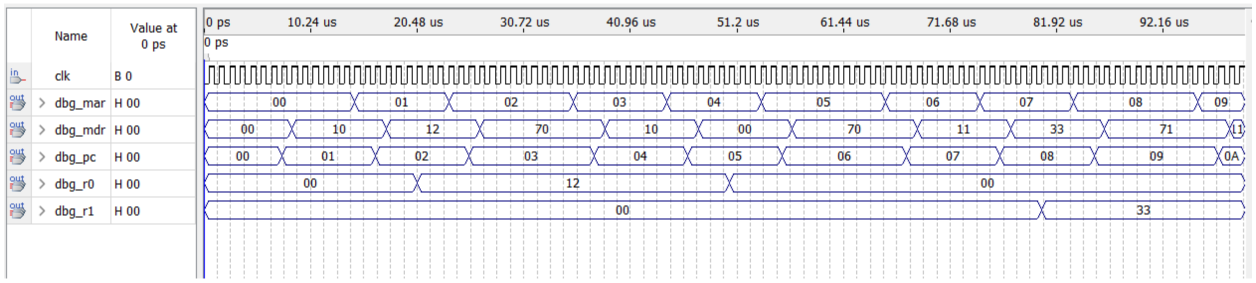


Figure 11. Waveform Test 7A (TESTNZ)

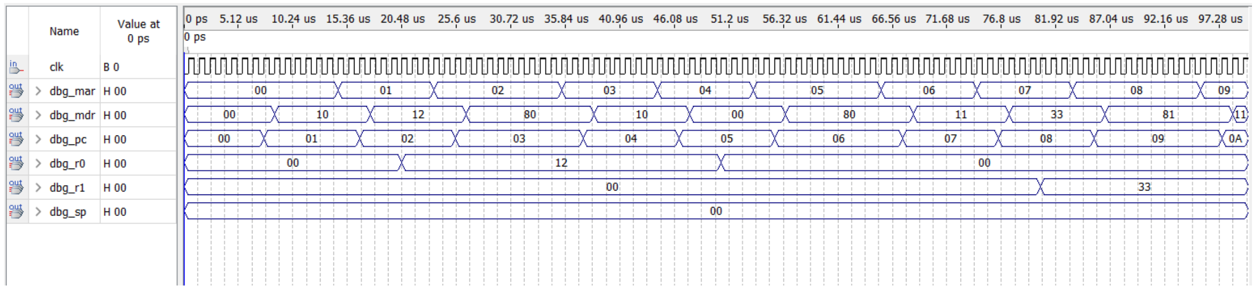


Figure 12. Waveform Test 7B (TESTZ)

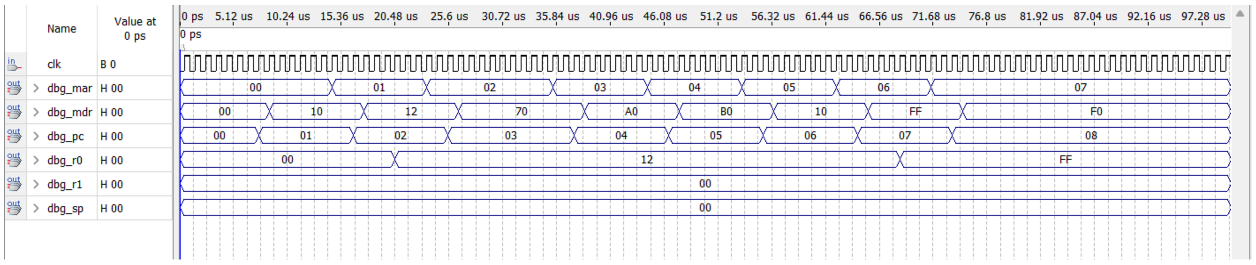


Figure 13. Waveform Test 8A (TESTNZ and JUMPZ)

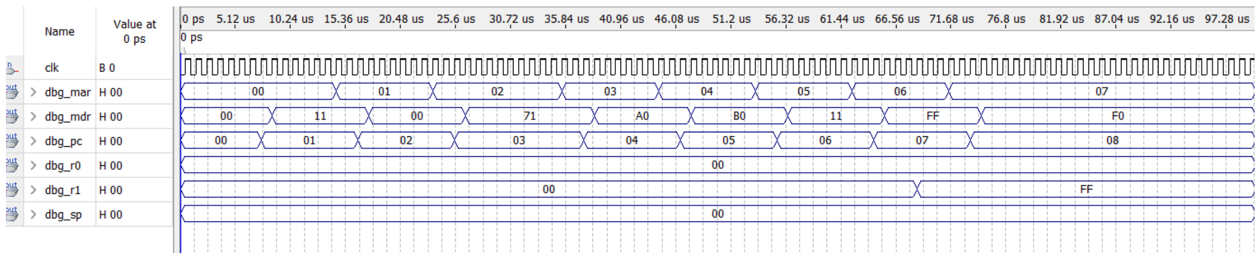


Figure 14. Waveform Test 8B (TESTNZ and JUMPZ) - Partially works

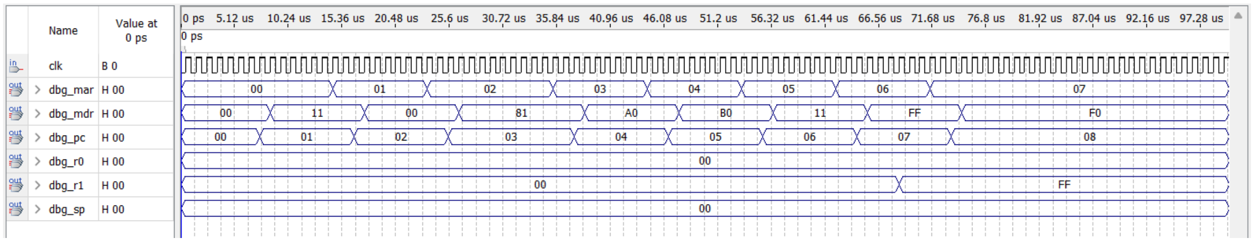


Figure 15. Waveform Test 9A (TESTZ and JUMPZ)

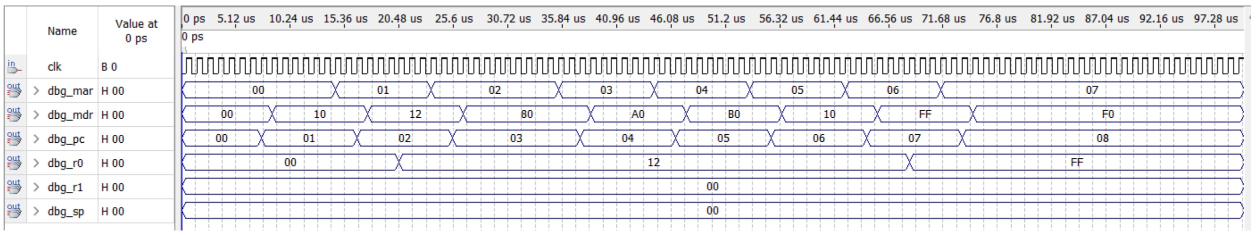


Figure 16. Waveform Test 9B (TESTZ and JUMPZ) - Partially works

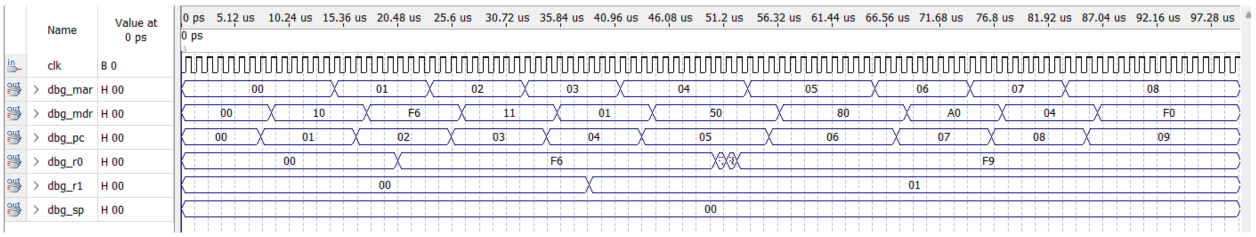


Figure 17. Waveform Test 10 (Loop) - Partially works

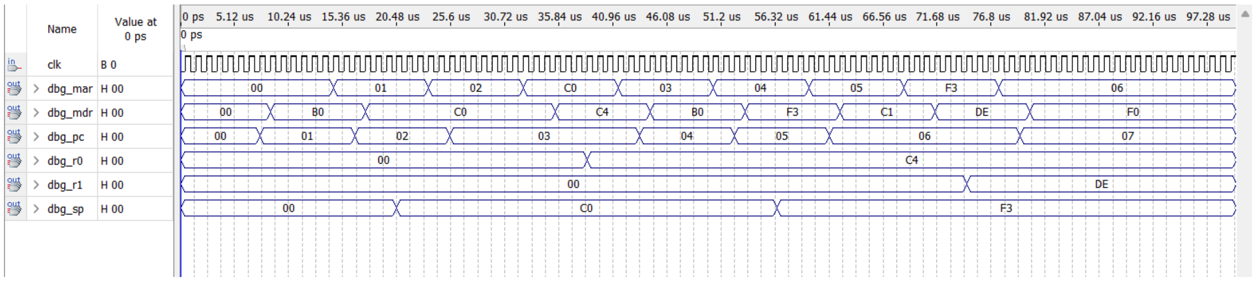


Figure 18. Waveform Test 11 (Stack)

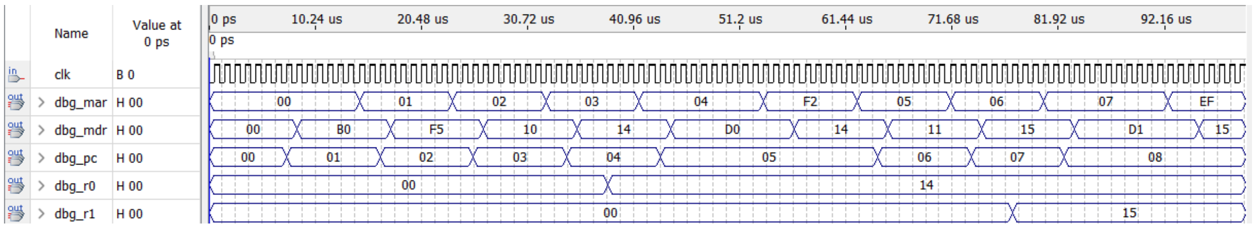


Figure 19. Waveform Test 12 (PUSH & POP)

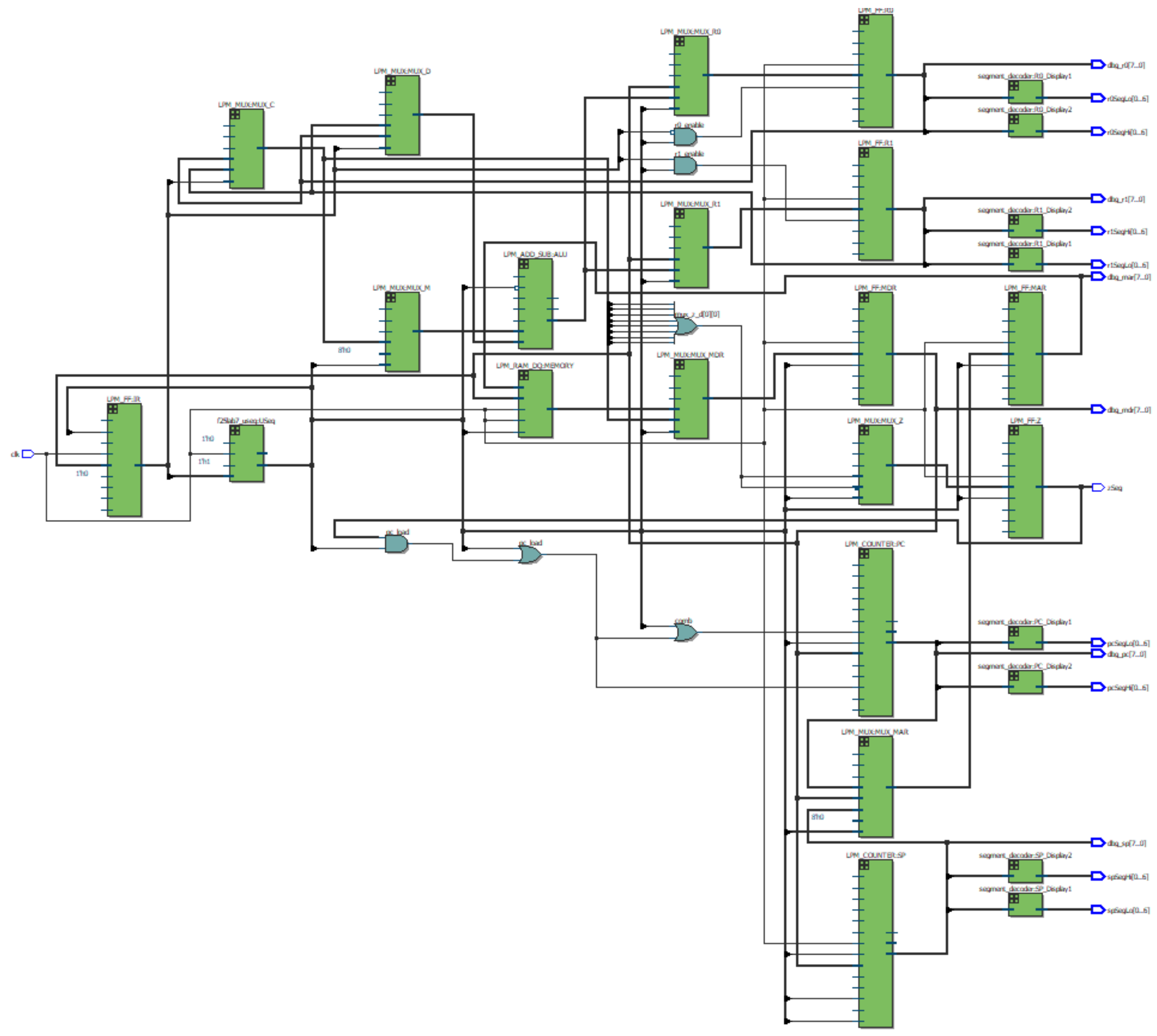


Figure 20. RTL Netlist Viewer

Results and Discussion

While testing our design on the board, we quickly discovered several issues. The first major problem involved the program counter. It was incrementing on every clock pulse, even though it should have incremented only once per asserted control signal. We resolved this by modifying the design so that the program counter incremented only on a change in the PC increment signal itself, which prevented multiple unintended increments. This was the primary obstacle we encountered while implementing the LOADI instruction.

We also ran into issues with the instruction register. Incorrect values were being loaded, causing non-instruction data to be interpreted as opcodes. As a result, the microsequencer occasionally treated the immediate data value of a LOADI instruction as a new instruction, leading to incorrect behavior. Our initial attempt to address this involved modifying the instruction register to use a similar activation method as the program counter, where it was enabled through a logical operation between the clock and the ir_load signal. While this appeared to work intermittently, it ultimately introduced a timing or race condition that made the behavior unreliable.

When we moved on to testing the LOAD instruction, additional issues surfaced. Specifically, we were unable to prevent the instruction register from loading data from the data bus instead of the intended instruction value. Due to time constraints during the lab session, we were unable to resolve this issue immediately. Afterward, we analyzed our design more carefully using the Quartus waveform simulator and reviewed both our code and micro-ROM contents.

Through this analysis, we discovered that an extra MAP signal was being asserted internally in both the FETCH2 and FETCH3 microstates, when it should only have been asserted during FETCH3. Additionally, we identified a more robust solution for the instruction register: disabling its synchronous clock input and instead using an asynchronous load enable controlled by the ir_load signal. This approach ensured that the instruction register always loaded the correct value, as long as the MDR held the instruction longer than the duration of the ir_load signal. This condition was satisfied because loading a new value into the MDR first requires updating the MAR, which naturally introduces a buffer cycle and prevents a race condition.

We also found that our microcode included an extra program counter increment during the LOAD instruction, which caused further issues. After correcting this, both the LOAD and STORE instructions functioned correctly.

However, we encountered more persistent challenges with the MOVE and ADD instructions. First, we discovered that the add/subtract control signal for the lpm_add_sub component was inverted, which resulted in incorrect arithmetic outputs. For the MOVE instruction, we reused the existing multiplexers designed for addition and subtraction by adding zero to one operand and transferring the result to the destination register, similar to how the mv pseudo-instruction operates in the RISC-V instruction set. This approach worked for the initial move operation when one register was empty. However, once both registers contained valid data, subsequent move operations caused a timing issue: although the correct register was initially overwritten, the adder updated its output on the following clock cycle while the destination register's enable signal was still active.

Unfortunately, we could not apply the same solution used for the instruction register in this case, because the input to the register was not static in the same way that the MDR is during instruction fetch. Based on our analysis, the only viable solution without extensively restructuring the design would be to introduce an accumulator register to act as a buffer. However, this would require a nontrivial change to the original architecture.

In retrospect, we believe our biggest mistake was not thoroughly testing our initial design using the Quartus waveform simulator before the lab session. Doing so would have allowed us to identify and address many of these issues earlier and more efficiently.

Full Lab 7 Zip Folder

Conclusion

Overall, we came out of this lab with a better understanding of how to design an entire CPU, going from RTL to block diagrams to the microcode ROM and finally to a complete VHDL implementation. Although we were unable to demonstrate all of the tests on time, we were able to modify the code afterward to pass most of the required tests. Most of the issues we encountered were not due to incorrect logic, but rather to control signals being asserted longer than intended. This led to unaccounted timing delays introduced by synchronous components, resulting in race conditions and unintended register updates that were difficult to diagnose without waveform analysis.

To improve our design in the future, we can ensure that all control signals are asserted for exactly one clock cycle, either through microcode changes or additional gating logic. Introducing a temporary accumulator register would simplify data movement operations and prevent timing conflicts when reusing the adder for move instructions. Besides, this would help us fix our problems with the synchronous loads, ensuring the correct registers are active when they should be.

Additionally, implementing a clear and consistent reset strategy, such as providing synchronous resets for the PC, SP, and microprogram counter, would reduce reliance on unknown initial states at startup. Finally, more extensive use of waveform simulation before hardware testing would allow timing-related issues to be identified earlier and significantly reduce debugging time during lab demonstrations.

References

1. ***F25 ECE495 L7.pdf***
2. ***F25 ECE495 L7 Part 3.pdf***
3. ***F25 ECE495 L7 Test.pdf***
4. ***F25 ECE495V12.pdf***
5. ***F25 ECE495V13.pdf***

Appendices

1. Full Code

```
library ieee; use ieee.std_logic_1164.all;
library lpm; use lpm.lpm_components.all;


entity lab7part3 is
      port(clk: in std_logic;
                dbg_pc  : out std_logic_vector(7 downto 0); --for waveform debugging
              dbg_r0  : out std_logic_vector(7 downto 0); --for waveform debugging
              dbg_r1  : out std_logic_vector(7 downto 0); --for waveform debugging
              dbg_mdr : out std_logic_vector(7 downto 0); --for waveform debugging
              dbg_mar : out std_logic_vector(7 downto 0); --for waveform debugging
              dbg_sp: out std_logic_vector(7 downto 0); --for waveform debugging
              r0SegLo, r0SegHi, r1SegLo, r1SegHi,
              pcSegLo, pcSegHi, spSegLo, spSegHi: out std_logic_vector(0 to 6);
              zSeg: out std_logic);

end lab7part3;

architecture structural of lab7part3 is


component f25lab7_useq is
  generic (uROM_width: integer := 10;
           uROM_file: string := "");
  port (opcode: in std_logic_vector(3 downto 0);
        uop: out std_logic_vector(1 to (uROM_width-9));
        debug_map_addr: out std_logic_vector(8 downto 0);  -- for debugging
        enable, clear: in std_logic;
        clock: in std_logic);
end component;

component segment_decoder is
    port (decoded_value: in std_logic_vector(3 downto 0);
            segment: out std_logic_vector(0 to 6));
end component;
```

```
-- MUX Input Signals
signal mux_mdr_d, mux_c_d, mux_d_d, mux_f_d, mux_m_d,
mux_r0_d, mux_r1_d: std_logic_2D(1 downto 0, 7 downto 0);

signal mux_mar_d: std_logic_2D(3 downto 0, 7 downto 0);

-- MUX Output Signals:
signal mux_mdr_q, mux_c_q, mux_d_q, mux_f_q, mux_m_q,
mux_r0_q, mux_r1_q, mux_mar_q: std_logic_vector(7 downto 0);


signal mux_z_q: std_logic_vector(0 downto 0);
signal mux_z_d: std_logic_2D(1 downto 0, 0 downto 0);

-- UOP Signal

signal uop: std_logic_vector(19 downto 0);

-- MUX Select Signals
signal mov_sel, im_sel, mdr_mux_sel, r_mux_sel, mar_mux_s1, mar_mux_s0, test_nz:
std_logic_vector(0 downto 0);


-- Increment/Decrement Signals
signal pc_inc, sp_up: std_logic;

-- Load Signals
signal sp_load, pc_load, mdr_load, mar_load, ir_load, write_z: std_logic;

-- Clear Signals
signal pc_clr: std_logic;

-- Flags and Counter
signal jumpz, inc_sp: std_logic;

-- Misc
signal add_or_sub, r_enable, mem_write,  next_addr: std_logic;

-- MEM
signal mem_q: std_logic_vector(7 downto 0);

-- Adder
signal sum: std_logic_vector(7 downto 0);

-- R0 and R1 Data In/Out
```

```
signal r0_enable, r1_enable: std_logic;
signal r0_data_out, r1_data_out: std_logic_vector(7 downto 0);

-- MAR/MDR Out
signal mar_q, mdr_q: std_logic_vector(7 downto 0);

--IR
signal IR_q: std_logic_vector(7 downto 0);

-- PC and SP
signal pc_q, sp_q: std_logic_vector(7 downto 0);

-- Z
signal z_q: std_logic_vector(0 downto 0);

signal temp_or, pc_temp_load: std_logic;

signal mar_mux_sel: std_logic_vector(1 downto 0);



begin

ALU: lpm_add_sub
        generic map(lpm_width => 8)
        port map(dataa => mux_m_q, datab => mux_d_q, result => sum, add_sub =>
add_or_sub);


MAR: lpm_ff
    generic map (lpm_width => 8)
    port map (data => mux_mar_q,
                q => mar_q,
                clock => clk,
                enable => mar_load);

R0: lpm_ff
    generic map (lpm_width => 8)
    port map (data => mux_r0_q,
                    q => r0_data_out,
                    clock => clk,
                    enable => r0_enable);
R1: lpm_ff
    generic map (lpm_width => 8)
        port map (data => mux_r1_q,
                        q => r1_data_out,
                        clock => clk,
                        enable => r1_enable);


MDR: lpm_ff
    generic map (lpm_width => 8)
    port map (data => mux_mdr_q,
                    q => mdr_q,
                    clock => clk,
                    enable => mdr_load);

PC: lpm_counter
            generic map (lpm_width => 8)
            port map (
                    clock => pc_inc or pc_load, --clk,
                    cnt_en => pc_inc,
                    sload => pc_load,
                    data => mdr_q,
                    q => pc_q


            );


SP: lpm_counter
            generic map (lpm_width => 8)
            port map (
                    clock => clk,
                    cnt_en => inc_sp,
                    sload => sp_load,
                    data => mdr_q,
                    updown => sp_up,
                    q => sp_q


            );



IR: lpm_ff generic map (lpm_width => 8)
    port map (data => mdr_q,
                    q => IR_q,
                    clock => clk, --clk,
                                         aload => ir_load,
                    enable => '0');

Z: lpm_ff
    generic map (lpm_width => 1)
    port map (data => mux_z_q,
                    q => z_q,
                    clock => clk,
                    enable => write_z);

MEMORY: lpm_ram_dq
        generic map (lpm_widthad => 8,
                    lpm_width => 8,
                    lpm_file => "lab7_ram11.mif")
        port map (address => mar_q,
                    data => mdr_q,
                    q => mem_q,
                    inclock => clk,
                    we => mem_write,
                    outclock => clk);

USeq: f25lab7_useq
    generic map (uROM_width => 29,
                 uROM_file => "ROM.mif")
        port map (opcode => IR_q(7 downto 4),
                    uop => uop,
                    enable => '1',
                    clear => '0',
                    clock => clk);




add_or_sub <= not uop(0);
write_z <= uop(1);
test_nz(0) <= uop(2);
ir_load <= uop(3);
mem_write <= uop(4);
mar_mux_s0(0) <= uop(5);
mar_mux_s1(0) <= uop(6);
mar_mux_sel <= mar_mux_s1 & mar_mux_s0;
mar_load <= uop(7);
r_mux_sel(0) <= uop(8);
r_enable <= uop(9);
r0_enable <= r_enable and (not IR_q(0));
r1_enable <= r_enable and IR_q(0);
mdr_mux_sel(0) <= uop(10);
mdr_load <= uop(11);
pc_temp_load <= uop(12);
pc_load <= pc_temp_load or (jumpz and z_q(0));
pc_inc <= uop(13);
inc_sp <= uop(14);
```

```
sp_load <= uop(15);
sp_up <= uop(16);
jumpz <= uop(17);
im_sel(0) <= uop(18);
mov_sel(0) <= uop(19);


MUX_MDR: lpm_mux
    generic map (lpm_size => 2,
                 lpm_width => 8,
                 lpm_widths => 1)
    port map (data => mux_mdr_d,
                sel => mdr_mux_sel,
                result => mux_mdr_q);

MUX_R0: lpm_mux
    generic map (lpm_size => 2,
                    lpm_width => 8,
                    lpm_widths => 1)
    port map (data => mux_r0_d,
                    sel => r_mux_sel,
                    result => mux_r0_q);

MUX_R1: lpm_mux
    generic map (lpm_size => 2,
                    lpm_width => 8,
                    lpm_widths => 1)
    port map (data => mux_r1_d,
                sel => r_mux_sel,
                result => mux_r1_q);

MUX_MAR: lpm_mux
    generic map (lpm_size => 4,
                    lpm_width => 8,
                    lpm_widths => 2)
    port map (data => mux_mar_d,
                    sel => mar_mux_sel,
                    result => mux_mar_q);

MUX_C: lpm_mux
    generic map (lpm_size => 2,
                    lpm_width => 8,
                    lpm_widths => 1)
    port map (data => mux_c_d,
                    sel => IR_q(0 downto 0),
                    result => mux_c_q);
```

```vhdl
MUX_D: lpm_mux
    generic map (lpm_size => 2,
                lpm_width => 8,
                lpm_widths => 1)
    port map (data => mux_d_d,
                sel => IR_q(0 downto 0),
                result => mux_d_q);


MUX_M: lpm_mux
    generic map (lpm_size => 2,
                lpm_width => 8,
                lpm_widths => 1)
    port map (data => mux_m_d,
                sel => mov_sel,
                result => mux_m_q);



temp_or <= mux_c_q(0) or mux_c_q(1) or mux_c_q(2) or mux_c_q(3) or mux_c_q(4) or
mux_c_q(5) or mux_c_q(6) or mux_c_q(7);

mux_z_d(0,0) <= temp_or;
mux_z_d(1,0) <= not temp_or;


MUX_Z: lpm_mux
    generic map (lpm_size => 2,
                lpm_width => 1,
                lpm_widths => 1)
    port map (data => mux_z_d,
                sel => test_nz,
                result => mux_z_q);

-- MUX Data Inputs
Gen_Statements: for i in 0 to 7 generate
        mux_mar_d(0, i) <= pc_q(i);
        mux_mar_d(1, i) <= mdr_q(i);
        mux_mar_d(2, i) <= sp_q(i);
        mux_mar_d(3, i) <= '0';

        mux_mdr_d(0, i) <= mem_q(i);
        mux_mdr_d(1, i) <= mux_c_q(i);

        mux_r0_d(0, i) <= mdr_q(i);
        mux_r0_d(1, i) <= sum(i);
        mux_r1_d(0, i) <= mdr_q(i);
        mux_r1_d(1, i) <= sum(i);

        mux_c_d(0, i) <= r0_data_out(i);
        mux_c_d(1, i) <= r1_data_out(i);

        mux_d_d(0, i) <= r1_data_out(i);
        mux_d_d(1, i) <= r0_data_out(i);

        mux_m_d(0, i) <= mux_c_q(i);
        mux_m_d(1, i) <= '0';
end generate;

R0_Display1: segment_decoder
       port map (decoded_value => r0_data_out(3 downto 0), segment => r0SegLo);
R0_Display2: segment_decoder
       port map (decoded_value => r0_data_out(7 downto 4), segment => r0SegHi);


R1_Display1: segment_decoder
      port map (decoded_value => r1_data_out(3 downto 0), segment => r1SegLo);
R1_Display2: segment_decoder
       port map (decoded_value => r1_data_out(7 downto 4), segment => r1SegHi);

PC_Display1: segment_decoder
       port map (decoded_value => pc_q(3 downto 0), segment => pcSegLo);
PC_Display2: segment_decoder
       port map (decoded_value => pc_q(7 downto 4), segment => pcSegHi);

SP_Display1: segment_decoder
       port map (decoded_value => sp_q(3 downto 0), segment => spSegLo);
SP_Display2: segment_decoder
       port map (decoded_value => sp_q(7 downto 4), segment => spSegHi);

zSeg <= z_q(0);

dbg_pc  <= pc_q;
dbg_r0  <= r0_data_out;
dbg_r1  <= r1_data_out;
dbg_mdr <= mdr_q;
dbg_mar <= mar_q;
dbg_sp <= sp_q;

end structural;
```

2. uSequencer Code

```vhdl
-- Fall 2025 ECE 495 Lab 7
-- uSequencer
-- Dr. Hou
--
library ieee;
use ieee.std_logic_1164.all;
library lpm;
use lpm.lpm_components.all;

entity f25lab7_useq is
  generic (uROM_width: integer := 10;
           uROM_file: string := "");
  port (opcode: in std_logic_vector(3 downto 0);
        uop: out std_logic_vector(1 to (uROM_width-9));
        debug_map_addr: out std_logic_vector(8 downto 0);  -- for debugging
        enable, clear: in std_logic;
        clock: in std_logic);
end f25lab7_useq;

architecture structural of f25lab7_useq is

signal uROM_address: std_logic_vector (7 downto 0);
signal uROM_out: std_logic_vector (uROM_width-1 downto 0);
signal uPC_mux_data: std_logic_2D(1 downto 0, 7 downto 0);
signal uPC_mux_sel: std_logic_vector(0 to 0);
signal uPC_mux_out: std_logic_vector(7 downto 0);
signal temp: std_logic_vector(7 downto 0);
begin
  temp <= opcode & "0000";
  for_label: for i in 0 to 7 generate
       uPC_mux_data(0, i) <= uROM_out(i);
       uPC_mux_data(1, i) <= temp(i);
  end generate;
  uPC_mux_sel(0) <= uROM_out(8);
  uPC_mux: lpm_mux
           generic map (lpm_width=>8, lpm_size=>2, lpm_widths=>1)
           port map (result=>uPC_mux_out, data=>uPC_mux_data, sel=>uPC_mux_sel);
  uPC: lpm_ff
           generic map (lpm_width=>8)
           port map (clock=>clock, data=>uPC_mux_out, q=>uROM_address,
                     sclr=>clear, enable=>enable);
  uROM: lpm_rom
            generic map (lpm_widthad=>8, lpm_width=>uROM_width,
lpm_file=>uROM_file)
            port map (address=>uROM_address, q=>uROM_out, inclock=>clock,
outclock=>clock);

  uop <= uROM_out(uROM_width-1 downto 9);
  debug_map_addr <= uROM_out(8 downto 0);
end structural;
```

3. Segment Decoder Code

```
library IEEE;
use IEEE.std_logic_1164.all;

entity segment_decoder is
    port (decoded_value: in std_logic_vector(3 downto 0);
            segment: out std_logic_vector(0 to 6));
end segment_decoder;

architecture behavioral of segment_decoder is

begin

  segment <= "0000001" when decoded_value = "0000" else
                 "1001111" when decoded_value = "0001" else
                 "0010010" when decoded_value = "0010" else
                 "0000110" when decoded_value = "0011" else
                 "1001100" when decoded_value = "0100" else
                 "0100100" when decoded_value = "0101" else
                 "0100000" when decoded_value = "0110" else
                 "0001111" when decoded_value = "0111" else
                 "0000000" when decoded_value = "1000" else
                 "0000100" when decoded_value = "1001" else
                 "0000100" when decoded_value = "1001" else
                 "0001000" when decoded_value = "1010" else
                 "1100000" when decoded_value = "1011" else
                 "0110001" when decoded_value = "1100" else
                 "1000010" when decoded_value = "1101" else
                 "0110000" when decoded_value = "1110" else
                 "0111000" when decoded_value = "1111" else
                 "1111111";
end behavioral;
```

4. uROM

```
WIDTH=29;
DEPTH=256;

ADDRESS_RADIX=HEX;
DATA_RADIX=BIN;

CONTENT BEGIN
    00: 00000000000000000000000000001;
    01: 00000000000010000000000000010;
    02: 00000010100000000000000000011;
    03: 00000000000000001000100000001;
    10: 00000000000010000000000010001;
    11: 00000010100000000000000010010;
    12: 01000000001000000000000000001;
    20: 00000000000010000000000100001;
    21: 00000010100000000000000100010;
    22: 00000000000010100000000100011;
    23: 00000001000100000000000100100;
    24: 00000000001000000000000000001;
    30: 00000000000010000000000110001;
    31: 00000010100000000000000110010;
    32: 00000000000010100000000110011;
    33: 00000001100000000000000110100;
    34: 00000000000000010000000000001;
    40: 10000000011000000000000000001;
    50: 00000000001100000000000000001;
    60: 00000000001100000001000000001;
    70: 00000000000000000110000000001;
    80: 00000000000000000010000000001;
    90: 00000000000010000000010010001;
    91: 00000010100000000000010010010;
    92: 00000001000000000000000000001;
    A0: 00000000000010000000010100001;
    A1: 00000010100000000000010100010;
    A2: 00100000000000000000000000001;
    B0: 00000000000010000000010110001;
    B1: 00000010100000000000010110010;
    B2: 00010000000000000000000000001;
    C0: 00000000000011000000011000001;
    C1: 00000000100000000000011000010;
    C2: 00000000001000000000000000001;
    D0: 00000100000000000000011010001;
    D1: 00000000000011000000011010010;
    D2: 00000000110000000000011010011;
    D3: 00000000000000010000000000001;
    E0: 00000000000011000000011100001;
    E1: 00000000100000000000011100010;
    E2: 00010100000000000000011100011;
    E3: 00000000001000000000000000001;
    F0: 00000000000000000000011110000;


END;
```

5. Pin Assignments

DE2 Signal Name	FPGA Pin No.	Description
pcSegHi[0]	PIN_AD18	Seven Segment Digit 5[0]
pcSegHi[1]	PIN_AC18	Seven Segment Digit 5[1]
pcSegHi[2]	PIN_AB18	Seven Segment Digit 5[2]
pcSegHi[3]	PIN_AH19	Seven Segment Digit 5[3]
pcSegHi[4]	PIN_AG19	Seven Segment Digit 5[4]
pcSegHi[5]	PIN_AF18	Seven Segment Digit 5[5]
pcSegHi[6]	PIN_AH18	Seven Segment Digit 5[6]
pcSegLo[0]	PIN_AB19	Seven Segment Digit 4[0]
pcSegLo[1]	PIN_AA19	Seven Segment Digit 4[1]
pcSegLo[2]	PIN_AG21	Seven Segment Digit 4[2]
pcSegLo[3]	PIN_AH21	Seven Segment Digit 4[3]
pcSegLo[4]	PIN_AE19	Seven Segment Digit 4[4]

pcSegLo[5]	PIN_AF19	Seven Segment Digit 4[5]
pcSegLo[6]	PIN_AE18	Seven Segment Digit 4[6]
r0SegHi[0]	PIN_M24	Seven Segment Digit 1[0]
r0SegHi[1]	PIN_Y22	Seven Segment Digit 1[1]
r0SegHi[2]	PIN_W21	Seven Segment Digit 1[2]
r0SegHi[3]	PIN_W22	Seven Segment Digit 1[3]
r0SegHi[4]	PIN_W25	Seven Segment Digit 1[4]
r0SegHi[5]	PIN_U23	Seven Segment Digit 1[5]
r0SegHi[6]	PIN_U24	Seven Segment Digit 1[6]
r0SegLo[0]	PIN_G18	Seven Segment Digit 0[0]
r0SegLo[1]	PIN_F22	Seven Segment Digit 0[1]
r0SegLo[2]	PIN_E17	Seven Segment Digit 0[2]
r0SegLo[3]	PIN_L26	Seven Segment Digit 0[3]
r0SegLo[4]	PIN_L25	Seven Segment Digit 0[4]
r0SegLo[5]	PIN_J22	Seven Segment Digit 0[5]
r0SegLo[6]	PIN_H22	Seven Segment Digit 0[6]
r1SegHi[0]	PIN_V21	Seven Segment Digit 3[0]
r1SegHi[1]	PIN_U21	Seven Segment Digit 3[1]
r1SegHi[2]	PIN_AB20	Seven Segment Digit 3[2]
r1SegHi[3]	PIN_AA21	Seven Segment Digit 3[3]
r1SegHi[4]	PIN_AD24	Seven Segment Digit 3[4]
r1SegHi[5]	PIN_AF23	Seven Segment Digit 3[5]

r1SegHi[6]	PIN_Y19	Seven Segment Digit 3[6]
r1SegLo[0]	PIN_AA25	Seven Segment Digit 2[0]
r1SegLo[1]	PIN_AA26	Seven Segment Digit 2[1]
r1SegLo[2]	PIN_Y25	Seven Segment Digit 2[2]
r1SegLo[3]	PIN_W26	Seven Segment Digit 2[3]
r1SegLo[4]	PIN_Y26	Seven Segment Digit 2[4]
r1SegLo[5]	PIN_W27	Seven Segment Digit 2[5]
r1SegLo[6]	PIN_W28	Seven Segment Digit 2[6]
spSegHi[0]	PIN_AD17	Seven Segment Digit 7[0]
spSegHi[1]	PIN_AE17	Seven Segment Digit 7[1]
spSegHi[2]	PIN_AG17	Seven Segment Digit 7[2]
spSegHi[3]	PIN_AH17	Seven Segment Digit 7[3]
spSegHi[4]	PIN_AF17	Seven Segment Digit 7[4]
spSegHi[5]	PIN_AG18	Seven Segment Digit 7[5]
spSegHi[6]	PIN_AA14	Seven Segment Digit 7[6]
spSegLo[0]	PIN_AA17	Seven Segment Digit 6[0]
spSegLo[1]	PIN_AB16	Seven Segment Digit 6[1]
spSegLo[2]	PIN_AA16	Seven Segment Digit 6[2]
spSegLo[3]	PIN_AB17	Seven Segment Digit 6[3]
spSegLo[4]	PIN_AB15	Seven Segment Digit 6[4]
spSegLo[5]	PIN_AA15	Seven Segment Digit 6[5]
spSegLo[6]	PIN_AC17	Seven Segment Digit 6[6]

zSeg	PIN_F17	LED Green[8]
sw1	PIN_AB28	Slide Switch[0]
sw2	PIN_AC28	Slide Switch[1]